2.5 TINY扫描程序的实现

现在开发扫描程序的真正代码以阐明本章前面所学到的概念。在这里将用到 TINY语言，它曾在第1章（1.7节）中被提到过，接着再讲一些由该扫描程序引出的实际问题。

2.5.1 为样本语言TINY实现一个扫描程序

第1章只是非常简要地介绍了一下TINY语言。现在的任务是完整地指出TINY的词法结构，也就是：定义记号和它们的特性。TINY的记号和记号类都列在表2-1中。

TINY的记号分为3个典型类型：保留字、特殊符号和“其他”记号。保留字一共有 8个，它们的含义类似（尽管直到很后面才需知道它们的语义）。特殊符号共有10种：分别是4种基本的整数运算符号、2种比较符号（等号和小于），以及括号、分号和赋值符号。除了赋值符号是两个字符的长度之外，其余均为一个字符。

表2-1 TINY语言的记号

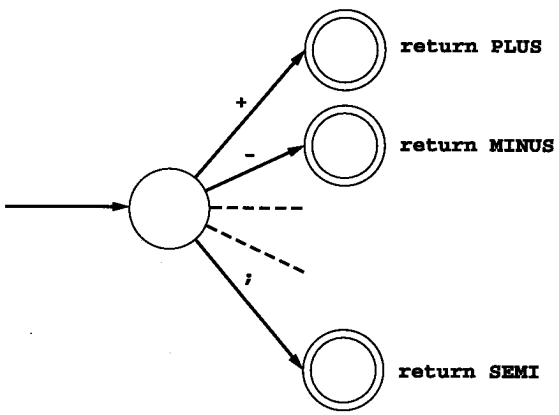
保 留 字	特 殊 符 号	其 他
if	+	**数**
then	-	（1个或更多的数字）
else	*	
end	/	
repeat	=	
until	<	**标识符**
read	(	(1个或更多的字母)
write	)	
	;	
	:=	

其他记号就是数了，它们是一个或多个数字以及标识符的序列，而标识符又是（为了简便）一个或多个字母的序列。

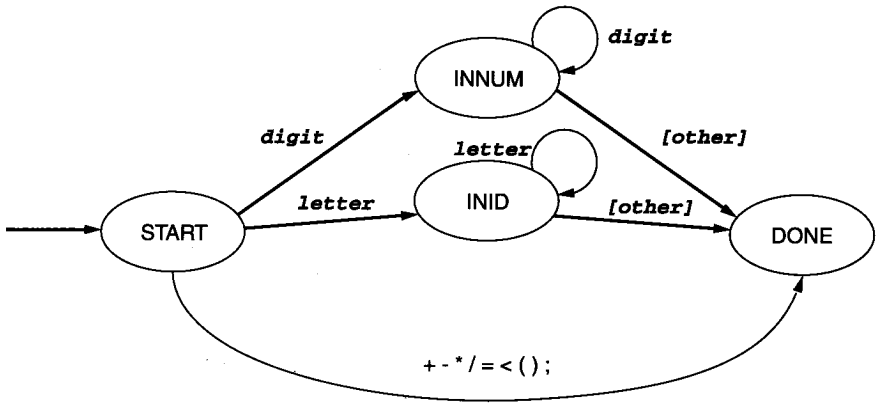
除了记号之外，TINY还要遵循以下的词法惯例：注释应放在花括号{...}中，且不可嵌套；代码应是自由格式；空白格由空格、制表位和新行组成；最长子串原则后须接识别记号。

在为该语言设计扫描程序时，可以从正则表达式开始并根据前一节中的算法来开发 NFA和DFA。实际上，前面已经给出了数、标识符和注释的正则表达式（TINY具有更为简单的版本）。其他记号的正则表达式都是固定串，因而均不重要。由于扫描程序的 DFA记号十分简单，所以无需按照这个例程就可直接开发这个DFA了。我们将按以下步骤进行。

首先要注意到除了赋值符号之外，其他所有的特殊符号都只有一个字符，这些符号的 DFA如下：



在该图中，不同的接受状态是由扫描程序返回的记号区分开来。如果在这个将要返回的记号（代码中的一个变量）中使用其他指示器，则所有接受状态都可集中为一个状态，称之为**DONE**。若将这个二状态的DFA与接受数和标识符的DFA合并在一起，就可得到下面的DFA：



请注意，利用方括号指出了不可被消耗的先行字符。

现在需要在这个DFA中添加注释、空白格和赋值。一个简单的从初始状态到其本身的循环要消耗空白格。注释要求一个额外的状态，它由花括号左边达到并在花括号右边返回到它。赋值也需要中间状态，它由分号上的初始状态达到。如果后面紧跟有一个等号，那么就会生成一个赋值记号。反之就不消耗下一个字符，且生成一个错误记号。实际上，未列在特殊符号中的所有单个字符既不是空白格或注释，也不是数字或字母，它们应被作为错误而接受，我们将它们与单个字符符号混合在一起。图2-8是为扫描程序给出的最后一个DFA。

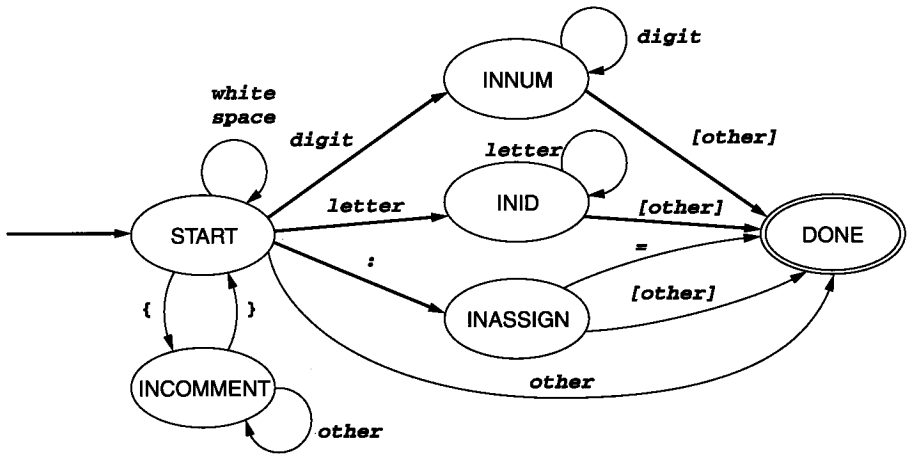


图2-8 TINY扫描程序的DFA

在上面的讨论或图2-8中的DFA都未包括保留字。这是因为根据DFA的观点，而认为保留字与标识符相同，以后再在接受后的保留字表格中寻找标识符是最简单的。当然，最长子串原则保证了扫描程序唯一需要改变的动作是被返回的记号。因而，仅在识别了标识符之后才考虑保留字。

现在再来讨论实现这个DFA的代码，它已被放在了**scan.h**文件和**scan.c**文件之中（参

见附录B)。其中最主要的过程是**getToken**（第674到第793行），它消耗输入字符并根据图2-8中的DFA返回下一个被识别的记号。这个实现利用了在2.3.3节中曾提到过的双重嵌套情况分析，以及一个有关状态的大型情况列表，在大列表中的是基于当前输入字符的单独列表。记号本身被定义成**globals.h**（第174行到第186行）中的枚举类型，它包括在表2-1中列出的所有记号以及内务记号**EOF**（当达到文件的末尾时）和**ERROR**（当遇到错误字符时）。扫描程序的状态也被定义为一个枚举类型，但它是位于扫描程序之中（第612行到第614行）。

扫描程序还需总地计算出每个记号的特性（如果有的话），并有时会采取其他动作（例如将标识符插入到符号表中）。在TINY扫描程序中，所要计算的唯一特性是词法或是被识别的记号的串值，它位于变量**tokenString**之中。这个变量同**getToken**一并是提供给编译器其他部分的唯一的两个服务，它们的定义已被收集在头文件 **scan.h**（第550行到第571行）。请读者注意声明了**tokenString**的长度固定为41，因此那个标识符也就不能超过40个字符（加上结尾的空字符）。后面还会提到这个限制。

扫描程序使用了3个全程变量：文件变量**source**和**listing**，在**globals.h**中声明且在**main.c**中被分配和初始化的整型变量**lineno**。

由**getToken**过程完成的额外的簿记如下所述：表**reservedWords**（第649行到第656行）和过程**reservedLookup**（第658行到第666行）完成位于由**getToken**的主要循环识别的标识符之后的保留字的查找，**currentToken**的值也随之改变。标志变量**save**被用作指示是否将一个字符增加到**tokenString**之上；由于需要包括空白格、注释和非消耗的先行，所以这些都是必要的。

到扫描程序的字符输入由**getNextChar**函数（第627行到第642行）提供，该函数将一个256-字符缓冲区内部的**lineBuf**中的字符取到扫描程序中。如果已经耗尽了这个缓冲区，且假设每一次都获取了一个新的源代码行（以及增加的 **lineno**），那么**getNextChar**就利用标准的C过程**fgets**从**source**文件更新该缓冲区。虽然这个假设允许了更简单的代码，但却不能正确地处理行的字数超过 255个字符的 TINY程序。在练习中，我们再探讨在这种情况下的**getNextChar**的行为（以及它更进一步的行为）。

最后，TINY中的数与标识符的识别要求从**INNUM**和**INID**到最终状态的转换都应是非消耗的（参见图2-8）。可以通过提供一个**ungetNextChar**过程（第644行到第647行）在输入缓冲区中反填一个字符来完成这一任务，但对于源行很长的程序而言，这也不是很好，练习将提到其他的方法。

作为TINY扫描程序行为的解释，读者可考虑一下程序清单2-3中TINY程序**sample.tny**（在第1章中已作为一个示例给出了）。程序清单2-4假设将这个程序作为输入，那么当**TraceScan**和**EchoSource**都是集合时，它列出了扫描程序的输出。

本节后面将详细讨论由这个扫描程序的实现所引出的一些问题。

程序清单2-3　TINY语言中的样本程序

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
```

```
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

程序清单2-4　当程序清单2-3中的TINY程序作为输入时，扫描程序的输出

```
TINY COMPILATION: sample.tny
    1: { Sample program
    2:   in TINY language -
    3:   computes factorial
    4: }
    5: read x; { input an integer }
        5: reserved word: read
        5: ID, name= x
        5: ;
    6: if 0 < x then { don't compute if x <= 0 }
        6: reserved word: if
        6: NUM, val= 0
        6: <
        6: ID, name= x
        6: reserved word: then
    7:   fact := 1;
        7: ID, name= fact
        7: :=
        7: NUM, val= 1
        7: ;
    8:   repeat
        8: reserved word: repeat
    9:     fact := fact * x;
        9: ID, name= fact
        9: :=
        9: ID, name= fact
        9: *
        9: ID, name= x
        9: ;
   10:     x := x - 1
       10: ID, name= x
       10: :=
       10: ID, name= x
       10: -
       10: NUM, val= 1
   11:   until x = 0;
       11: reserved word: until
       11: ID, name= x
       11: =
       11: NUM, val= 0
       11: ;
   12:   write fact { output factorial of x }
       12: reserved word: write
       12: ID, name= fact
   13: end
       13: reserved word: end
       14: EOF
```

2.5.2 保留字与标识符

TINY对保留字的识别是通过首先将它们看作是标识符，之后再在保留字表中查找它们来

完成的。这在扫描程序中很平常，但它却意味着扫描程序的效率须依赖于在保留字表中查找过程的效率。我们的扫描程序使用了一种非常简便的方法——线性搜索，即按顺序从开头到结尾搜索表格。这对于小型表格不成问题，例如TINY中的表格，它只有8个保留字，但对于真实语言而言，这却是不可接受的，因为它通常有30~60个保留字。这时就需要一个更快的查找，而这又要求使用更好的数据结构而不是线性列表。假若保留字列表是按字母表的顺序写出的，那么就可以使用二分搜索。另一种选择是使用杂凑表，此时我们希望利用一个冲突性很小的杂凑函数。由于保留字不会改变（至少不会很快地），所以可事先开发出这样一个杂凑函数，它们在表格中的位置对于编译器的每一步运行而言都是固定的。人们已经确定了各种语言的最小完善杂凑函数（minimal perfect hash function），也就是说能够区分出保留字且具有最小数值的函数，因此杂凑表可以不大于保留字的数目。例如，如果只有8个保留字，则最小完善杂凑函数总会生成一个0~7的值，且每个保留字也会生成不同的值（参见“注意与参考”一节）。

在处理保留字时，另一个选择是使用储存标识符的表格，即：符号表。在过程开始之前，将所有的保留字整个输入到该表中并且标上“保留”（因此不允许重新定义）。这样做的好处在于只要求一个查找表。但在TINY扫描程序中，直到扫描阶段之后才构造符号表，因此这个方法对于这种类型的设计并不合适。

2.5.3 为标识符分配空间

TINY扫描程序设计中的另一个缺点是记号串最长仅为40个字符。由于大多数的记号的大小都是固定的，所以对于它们而言这并不是问题；但是对于标识符来讲就麻烦了，这是因为程序设计语言经常要求程序中的标识符长度为任意值。更糟的是：如果为每一个标识符都分配一个40个字符长度的数组，那么就会浪费掉大多数的空间；这是因为绝大多数的标识符都很短。由于使用了实用程序函数**copyString**复制记号串，其中copyString函数动态地分配仅为所需的空间（如同将在第4章看到的一样），TINY编译器的代码就不会出现这个问题了。**TokenString**长度限制的解决办法与之类似：仅仅基于需要来分配，有可能使用**realloc**标准C函数。另一种办法是为所有的标识符分配最初的大型数组，接着再在该数组中按照自己做的方式进行存储器的分配（这是将在第7章要谈到的标准动态存储器管理计划的特殊情况）。